



Treball Final de Grau

GRAU D'ENGINYERIA INFORMÀTICA

**Facultat de Matemàtiques i Informàtica
Universitat de Barcelona**

**Q-LEARNING IN COLLABORATIVE MULTIAGENT
SYSTEMS**

Alfred Gonzalez Trastoy

Director: Maite López-Sánchez
At: Departament de Matemàtiques i Informàtica
Barcelona, February 2018

ACKNOWLEDGEMENTS

Before any introduction, I would like to thank my project director at Universitat de Barcelona, Maite López-Sánchez, for providing me with an unpublished version of the winner of the Malmo Collaborative AI Challenge (MCAC), which inspired me to start this project to try and solve the requested problem, and for guiding me when things were not working as planned. I also would like to thank Microsoft for encouraging students around the world to try to solve challenging tasks in collaborative AI with projects like the mentioned MCAC.

My gratitude to all the professors I had over my career for inspiring me and others to get interested on many unknown subjects and to widen my view point on the already-known ones.

Lastly but not least, thanks to my family, friends and workmates for your support, friendship and help over the years.

Thanks.

ABSTRACT

Q-learning is one of the most widely used reinforcement learning techniques. It is very effective for learning an optimal policy in any finite Markov decision process (MDP). Collaborative multiagent systems, though, are a challenge for self-interested agent implementation, as higher utility can be achieved via collaboration. To evaluate the Q-learning efficiency in collaborative multiagent systems, we will use a simplified version of the Malmo Collaborative AI Challenge (MCAC). It was designed by Microsoft and consists of a game where 2 players can collaborate to catch the pig (high reward) or leave the game (low reward). Each action costs 1, so knowing when to leave and when to chase the pig is key for achieving high scores. Two main problems are faced in the challenge: uncertainty of the other agent behaviour and a limited learning time. We propose solutions to both problems using a simplified MCAC environment, a state-action abstraction and an agent type modelling. We have implemented an agent that is able to identify the other player behaviour (whether it is collaborating or not) and can learn an optimal policy against each type of player. Results show that Q-learning is an efficient and effective technique to solve collaborative multiagent systems.

INDEX

1. Introduction	6
1.1 Motivation	6
1.2 Technology	7
1.2.1 Malmo project	7
1.2.2 The Malmo Collaborative AI Challenge (MCAC)	7
1.3 Objectives	8
1.4 Main problems	9
1.5 Project structure	9
2. Game characteristics	10
3. A brief introduction to Q-learning and collaborative multiagent systems	12
3.1 Multiagent systems (MAS)	12
3.2 Markov decision process (MDP)	12
3.3 Reinforcement learning (RL)	12
3.4 Q-learning	13
4. Development: design and implementation	15
4.1 Approach of our design	15
4.1.1 First problem approach	15
a) A simplification of the MCAC terrain	15
b) A state-action abstraction	16
4.1.2 Second problem approach	17
a) An agent behaviour modelling	17
4.2 Q-learning implementation	17
4.2.1 Q-learning training algorithm	18
4.2.2 Q-learning testing algorithm	19
5. Experiments and results	20
6. Conclusions	23
7. References	24
8. Annexes	26
8.1 Previous installation	26
8.2 Running the original Malmo Collaborative AI Challenge (MCAC)	26
8.3 My code	26
8.4 Running my agent	27

1. Introduction

1.1 Motivation

Any environment, like university, work, hospitals, animal behaviour... can be modelled [1], through a Markov decision process (MDP), into a multiagent system (MAS), where agents interact among themselves. That's why researchers of AI are so interested in developing solutions to MAS, as we will explain now.

[1] "Many problems such as (stochastic) planning problems, learning robot control and game playing problems have successfully been modelled in terms of an MDP. In fact MDPs have become the de facto standard formalism for learning sequential decision making." Extract from [2]

Many big companies have invested in AI projects as a way to solve zero-sum games like chess, Go, checkers... and even some Atari 2600 games, among others, with great success. We have the famous Deep Blue by IBM, that was able to win the former world champion Garry Kasparov at 1997 [3]; an Atari project from Google DeepMind [4] that, using a variant of Q-learning, surpasses human expert level at some Atari 2600 games¹ [5]; AlphaGo and AlphaGo Zero [6] from Google DeepMind, that uses reinforcement learning on Go game, was able to defeat the world's best players and number one *Ke Jie*² [7] and its recent AlphaZero [8] with a tabula rasa reinforcement learning, performing at superhuman level in many challenging domains (chess, shogi and Go)³ [9].

In this context, with this project we have worked over a collaborative multiagent system, where agents can either work together to achieve a common goal (high reward) or focus on their individual goals (low reward). Further research in collaborative AI will provide a way to develop more and more complex machines capable of collaborating with other machines and humans to achieve their goals. Being able to teach agents how to recognize others' intentions (whether they might be collaborating or not) and what

¹ *"The model is a convolutional neural network, trained with a variant of Q-learning, (...). We apply our method to seven Atari 2600 games from the Arcade Learning Environment, (...). We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them."* [5]

² *"After just three days of self-play training, AlphaGo Zero emphatically defeated the previously published version of AlphaGo - which had itself defeated 18-time world champion Lee Sedol - by 100 games to 0. After 40 days of self training, AlphaGo Zero became even stronger, outperforming the version of AlphaGo known as "Master", which has defeated the world's best players and world number one Ke Jie."* [7]

³ *"Starting from random play, and given no domain knowledge except the game rules, AlphaZero achieved within 24 hours a superhuman level of play in the games of chess and shogi (Japanese chess) as well as Go, and convincingly defeated a world-champion program in each case."* [9]

actions are optimal in each case (when and how to work together or go solo), is one of the biggest problems in collaborative AI.

The main objective in collaborative AI is to teach an agent how to adapt, instantly, to environment changes and other agents' behaviours. However, the sequential decision-making in collaborative MAS faces two key problems: the large state space and the uncertainty of the other agents' behaviour.

To solve those problems, in this project, we will use a type belief modelling, to identify the agents' behaviour, and a state-action abstraction on top of one of the most extended techniques to solve collaborative MAS, Q-learning. With Q-learning, we can teach an agent the optimal action on each state based on the rewards acquired on previous iterations. Q-learning has been proved to converge to the optimum action-values [10].

1.2 Technology

We chose the Malmo Collaborative AI Challenge (MCAC) as it offers a complex domain where we can design and implement an agent over a collaborative multiagent system.

1.2.1 Malmo project

The Malmo project [11], which is built on top of the popular multiplayer game Minecraft, is a platform for Artificial Intelligence experimentation and research.

1.2.2 The Malmo Collaborative AI Challenge (MCAC)

The MCAC [12] is designed over the Malmo project and aims to encourage the development of agents that can learn an optimal policy in collaborative multiagent systems. It is inspired by a variant of the stag hunt by Yoshida et. al [13]. The stag hunt is a classical "game theoretic" game formulation that describes the conflict between social collaboration and individual safety. It consists of two players that can either work together to try to catch the stag (high reward) or deviate from collaboration and catch the hare (low reward).

The MCAC consists of a mini-game in which agents can work together to achieve a common goal (catch the pig) or deviate from collaboration (go to exit). The idea of the challenge is to develop an agent that can learn how to perform well (achieve high scores) while playing alongside different types of agents. It will have to learn which action to take on each state, depending on the other agents'.

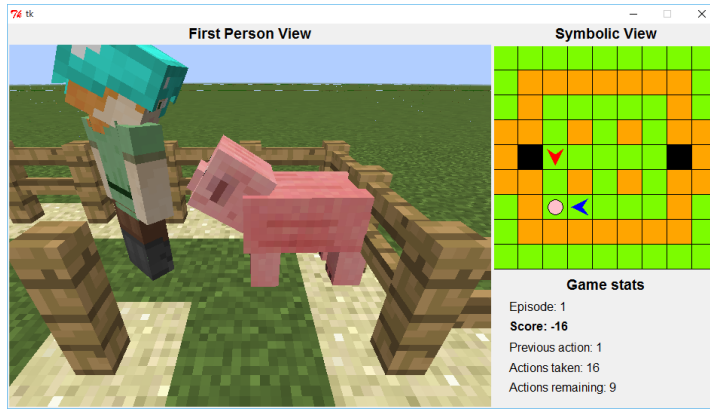


Figure 1. The MCAC representation of the variant of the stag hunt. Players (red and blue) can trap the pig (pink) or go to the exit (black).

The mini-game is characterized by being a win-win collaborative game, where players can achieve high scores by collaborating, unlike many other games (chess, checkers, tic-tac-toe...) where the gain from one player is balanced with the loss of another. Following the classification of environment properties [14], we can define our environment as being:

- Partially accessible: partially observable information of the environment.
- Non-deterministic: we do not know the future state after performing an action, as there are other agents that can perform many actions.
- Discrete: finite number of actions and states.
- Dynamic: it is constantly changing by the actions of other agents.
- Sequential: the current decision can affect all future decisions.

1.3 Objectives

The main goal of this project is to develop an agent that can learn how to improve the performance (achieve high scores) in a simplified version of the MCAC collaborative multiagent system. We apply Q-learning to implement our agent so that it learns the optimal policy for each agent type (collaborative or not) and uses a type belief modelling, which will let us identify when is the other agent collaborating and when is focused on other tasks. To reduce the learning time of the algorithm we propose a state-action abstraction.

Through this project, we are proving the validity of Q-learning in collaborative MAS. After having the complete system, we observe and measure the improvement of our agent: the more training the better the mean score should be. The validity of the entire algorithm and previous statement is tested and displayed in a learning curve (score-wise) chart. Other results are used to theorize about the possible interactions between our agent and the MCAC environment.

1.4 Main problems

In this project, we face two main problems

- The Minecraft platform is extremely time consuming. It usually takes around 3-5 seconds to execute a single episode (game), so training for 100.000 episodes would take around 4-5 days. Having a large state-action space means that to learn an effective policy would take too long.
- We have uncertainty of the other agents' behaviour.

To solve these problems, we propose:

- A simplification of the MCAC world.
 - The board has been reduced.
 - A state-action abstraction (inspired by the solution of the winner of the MCAC [15]).
- A modelling of the agent behaviour.

1.5 Project structure

The project will be presented with the following points:

- Game characteristics: to explain the rules and interactions in the MCAC pig chase game.
- A brief introduction to Q-learning and collaborative multiagent systems: to introduce the basics of a Q-learning algorithm.
- Development: design and implementation of our agent. Explanation of the approach design and implementation of the Q-learning.
- Experiments and results
- Conclusions

2. Game characteristics

The original MCAC game and environment has the following characteristics. We will present later our simplified version with some changes.

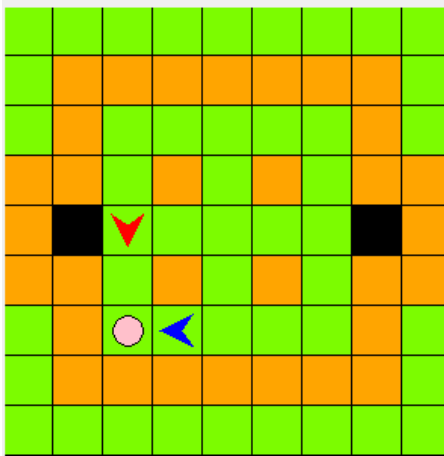


Figure 2. That is the original MCAC abstract representation of the game. We can see Agent 1 (blue), Agent 2 (red), Pig (pink), two exits (black) and the impassable terrain (orange).

- **Terrain:** 9x9 grid with
 - Some impassable terrain (orange)
 - Two exits (black)
- **Agents:** there are 2 players and 1 pig
 - Pig (pink)
 - Defined by position (x, y)
 - 4 move actions to adjacent squares: north, south, east, west
 - Players
 - Defined by position and direction (x, y and direction)
 - 3 available actions:
 - Rotate (right or left)
 - Move forward
 - Player 1 (the other player, blue): its behaviour is selected at the start of the episode (game) with some probability:
 - 25% of being random: it takes random actions.
 - 75% of being focused: it chases the pig, taking the shortest path.
 - Player 2 (red): our agent
- **Rules**
 - Agents cannot move into impassable terrain
 - If players are facing into impassable terrain they will only have 2 possible actions: rotate right or left.
 - Every step (turn), players perform an action.
 - The pig can perform many actions at a time, or none.

- An episode (game) consists of many steps and ends when happens any of the next:
 - The pig has been captured: surrounded by impassable terrain and agent/s in its 4 adjacent squares (see *Figure 3*).
 - Maximum steps done (25).
 - A player reaches the exit.
- Rewards (per step)
 - Each player action costs 1
 - Capturing the pig gives 25
 - Getting to an exit gives 5
- The game score is the sum of the rewards of all the steps in a single episode. It ranges from -25 to 25.

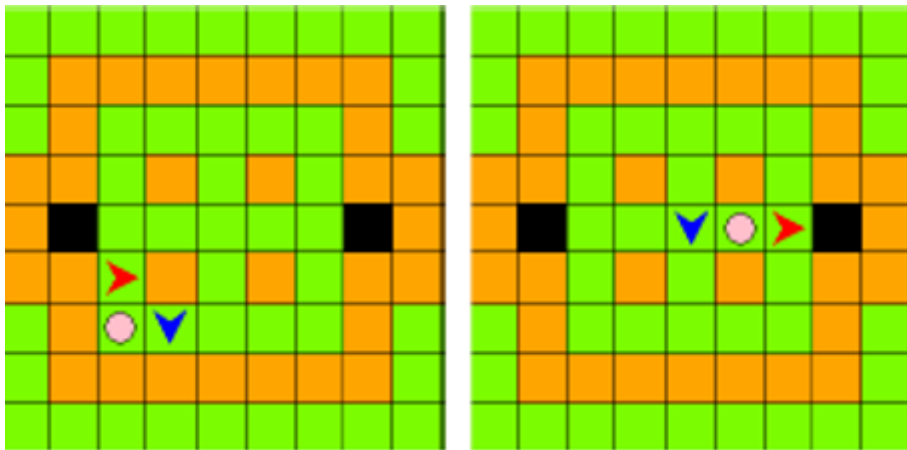


Figure 3. Two different examples of a way to capture the pig.

3. A brief introduction to Q-learning and collaborative multiagent systems

3.1 Multiagent systems (MAS)

A **multiagent system (MAS)** consists of multiple agents interacting within an environment. Each agent has its own goals (self-interested) and they might cooperate or not. In a collaborative MAS, agents might collaborate to perform a task (high score) or focus on their individual goals (low score). Finding a way to develop an agent that can make optimal sequential decisions is a challenge: we need to look at long term rewards.

3.2 Markov decision process (MDP)

[1] “Many problems such as (stochastic) planning problems, learning robot control and game playing problems have successfully been modelled in terms of an MDP. In fact, MDPs have become the *de facto* standard formalism for learning sequential decision making.” Extract from [2]

We use **Markov decision process (MDP)**, a mathematical framework, for modelling decision making [2]. We will explain what it consists of by using as examples the game characteristics presented previously:

- Set of states (S): all the combination of possible positions and directions of the players and the pig positions at each step.
- Set of actions (A): available actions to perform (move forward, turn left and turn right).
- Reward function $R(s,a)$: a function that returns the utility of an action a at a given state s .
 - $R: S \times A \rightarrow R$
 - It is like the reward system of -1 for moving and +25 and +5 for catching the pig and going to the exit, respectively.

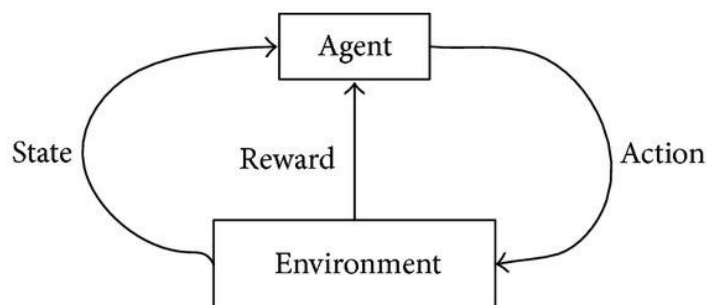


Figure 4. After performing an action at a given state, we can observe the reward and future state. Image from [16].

3.3 Reinforcement learning (RL)

Reinforcement learning is a technique that deals with learning sequential decision-making problems [2].

It is based on behaviourist psychology, teaching the algorithm which actions to take on each state by a rewarding and punishing system. Similar to dog training.

3.4 Q-learning

Q-learning [17] is a reinforcement learning technique that can find an optimal policy in any finite Markov decision process (MDP). The idea behind the algorithm is to teach an agent how to act on each state by using the previous experiences (rewards), that can be either positive or negative. It consists of

- States (S): a set of all possible states.
- Actions (A): a set of available actions.
- Rewards (r): utility after taking action a at state s .
- **Q-function**: a state-action function that gives the expected utility of taking action a at state s : $Q(s,a)$
 - $Q: S \times A \rightarrow Real$
 - Acts like a table that keeps the expected value for all the state-action pairs.
- A **policy** (π) maps the action to take at any state s . It returns the action with the highest Q-value at each state:
 - $\pi(s) = A$
 - $A^*: \text{argmax}(Q(s,a))$
 - Agent checks the q-values of each action at state s and selects the one with higher utility.
- **Exploration vs exploitation policy**: balancing the ratio between exploration and exploitation of actions can be a difficult task. Too much exploration causes the learning to be too random and might not be able to find good policies. Too much exploitation may prevent the agent from finding other potentially better actions. It is known as the dilemma of the exploration vs exploitation [18]. ϵ -greedy method [19] has good performance and it is usually the first choice [20].
- **ϵ -greedy policy** [19] (ϵ): while learning, we may choose an off-policy action with probability ϵ to explore other options.
 - $1 - \epsilon$ probability to take the action returned by the policy (the action with the highest Q-value)
 - ϵ probability to take a random action (off-policy)
 - $0 < \epsilon < 1$
- **Learning rate** (α): when updating the Q-values, we must determine the rate at which the Q-function will be learning. It gives some weights to the old q-value and the new reward.
 - New q-value = $(1 - \alpha) * \text{old_q_value} + \alpha * \text{new_reward}$
 - If $\alpha = 0$ it won't learn anything and if $\alpha = 1$ the agent would only use the new reward.
 - Usually fixed at 0.1

Q-learning is divided in two phases:

- **1st the training**, where the agent will learn the optimal actions at each state. Looking at *Figure 5*, for each iteration, the algorithm choses an action with the ϵ -greedy policy, performs the action and observes the consequent reward and state. After each step, the Q-values of all the visited states are updated. It ends when a terminal state is reached, that is when the episode is finished. We repeat the process for the desired number of training episodes we want for the agent.

```
Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ ;
  until  $s$  is terminal
```

Figure 5. Q-learning algorithm. Image extracted from Sutton and Barto [18].

- **2nd the testing**, where the agent will use the learnt policy. At each state, it will take the action with the highest Q-value. After many training iterations, the efficiency of the Q-learning can be observed. The more training the better the results will be, converging into the optimum action-values [1].

Therefore, we will model our collaborative multiagent system (MAS) as a Markov decision process (MDP) and use Q-learning to find an optimal policy.

4. Development: design and implementation

A reminder of the problems we are facing:

- Learning an effective policy is extremely **time consuming** in the Minecraft platform. Having a **large state-action space** increases the time it takes to effectively train the algorithm.
- There is **uncertainty** of the other agent type (random or focused)

4.1 Approach of our design

4.1.1 First problem approach

To solve the first problem, two solutions are proposed.

a) A simplification of the MCAC terrain

A simplified version of the MCAC terrain is one of the solution approaches: having less possible states (positions and steps) means that the algorithm won't need as much time to converge into optimum action-values. It consists of the following modifications:

- The board has been reduced to a 6x6 grid from the original 9x9 grid (see *Figure 6*).
 - 9 possible positions
- It has only 1 exit.
- The maximum number of steps has been reduced from 25 to 15
- The rewarding system has been modified to adapt to the changes:
 - Catching the pig return value goes from 25 to 15.
 - Since the maximum distance only decreases by 1 (from 5 to 4), it was not necessary to modify its reward.

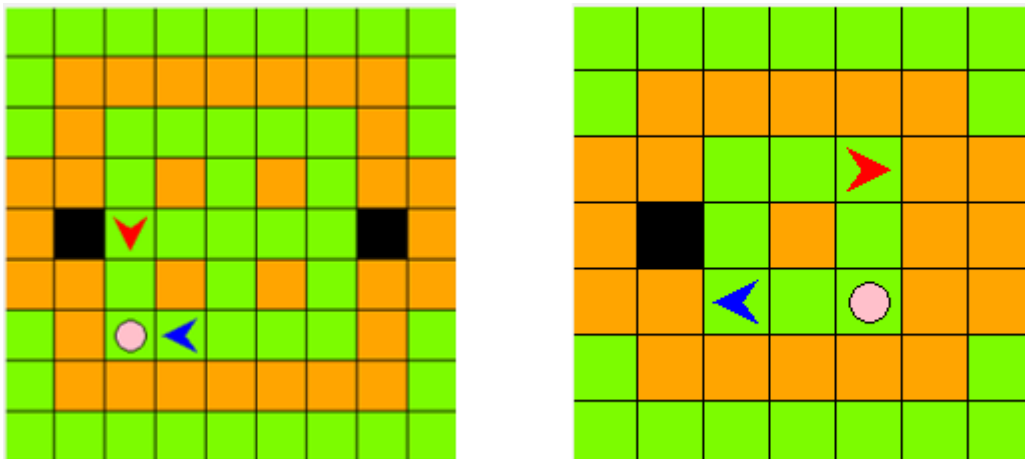


Figure 6. Original MCAC board (left) and our adaptation (right).

This way we go from 23 positions and 25 maximum steps to 9 positions and 15 steps. And the rewarding system is decreased to adapt to these changes, lowering the pig return value from 25 to 15.

b) A state-action abstraction

(Inspired by the solution of the MCAC winner [11])

There are 9 possible positions for each agent and players have 4 directions (unlike the pig), so the combination of players and pig positions are $9 \times 4 \times 9 \times 4 \times 9 = 11.664$. To that we have to add the 15 steps of an episode. The possible number of states is then $11.664 \times 15 = 174.960$. Moreover, there are 3 possible actions and so the state-action space is 524.880. Taking into consideration the time it takes to complete an episode (approximately 4 seconds), it would take too long to train it: with a mean of 6 steps per episode it would take 4 days to step once on each possible state-action.

Because of that, reducing the state-action space was required: we made a state-action abstraction. The relative positions (distances) between agents, pig and exit are taken into consideration for the state space, instead of the absolute positions. The action space is reduced from 3 (move, turn left and turn right) to 2 possibilities: chase the pig or go to exit. The final state-action abstraction consists of 3 possible pig states, 5 possible distances from player 1 to pig and another 5 from player 2 to pig, 5 possible distances from player 2 (our agent) to exit, 15 possible steps and 2 possible action.

- **State-action abstraction space $\rightarrow 3 \times 5 \times 5 \times 5 \times 15 \times 2 = 11.250$ possibilities**
 - Pig state (0-2): if catchable by no one, 1 or 2 agents.
 - Not catchable (uncatchable position) $\rightarrow 0$
 - Only 1 position in the board (at *Figure 7*, the red circle).
 - Catchable by 1 agent $\rightarrow 1$
 - Only 1 position too (at *Figure 7*, the yellow circle).
 - Catchable by 2 agents $\rightarrow 2$
 - The other 7 positions (at *Figure 7*, the blue circle).

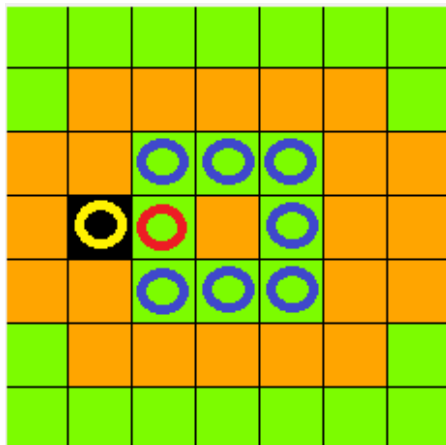


Figure 7. In red the position where the pig is uncachable, in yellow where it is catchable by a single agent and in blue where it is only catchable by the collaboration of 2 agents.

- Agent 1 distance to pig (0-4): we use Manhattan distances [21].
- Agent 2 distance to pig (0-4)
- Agent 2 distance to exit (0-4)

- Number of steps made (1-15)
- Action to perform (0-1)
 - Head to exit → 0
 - Chase pig → 1

4.1.2 Second problem approach

To solve the problem of the uncertainty of the other agent type, we use a modelling of the agent 1 behaviour:

a) An agent behaviour modelling

Agent 1 type is initialized at random or focused at the beginning of each episode, using the values proportioned by the MCAC characteristics, with a probability of:

- 25% Random
- 75% Focused

After the initialization, we have to keep updating the type belief of the agent 1 on each iteration. To do that, we use the agent 1 and pig position on the previous step and the agent 1 actual position.

- On each step, update the type:
 - If it did not try to get closer to the pig on the previous step, we know it is random and we won't need to update the agent 1 type on any of the next iterations.
 - If distance is lower, it can either be random or focused, since a random agent can get closer too. We will consider it being focused but will keep updating the agent 1 type, until we find it is random or until the end of the episode.

The uncertainty in the last case can be ignored with a low risk of it affecting the overall performance. In the worst-case scenario, agent 1 have two possible optimal actions at first step. It can either turn right or turn left if it is facing to the opposite direction of the pig or if there are two possible paths to catch it. Therefore, to make a whole turn and move forward while being random would be at a probability of

- $2/3 * 1/3 * 1/3 = 7,4\%$
- After only 3 steps we are sure with a >92% certainty of it being focused, >97% at step 4 and >99% at step 5.

Even if it is a random agent after 2-3 steps, the agents might be able to catch the pig anyways. It is a feature that our agent will be able to learn on the Q-learning.

4.2 Q-learning implementation

- 2 Q-functions are used in our implementation, one for each agent type.

- Q-Random: stores the Q-values of all the state-action pairs trained against a Random agent.
- Q-Focused: stores the Q-values against a Focused agent.
- Depending on the agent 1 type belief, we will use one function or the other, getting the expected utility of action a at state s against the agent type we are facing.
- **ϵ -greedy policy ratio**
 - An $\epsilon = 0.1$ is often used for any Q-learning algorithm, as it ensures some exploration to find potential optimal actions while being able to exploit the actual optimal actions.
 - We have a 10% probability to make a non-optimal action (off-policy). As there are only two actions it will chose the other possible action.
- **Learning rate**
 - We must choose a learning rate that guarantees a proper incorporation of the new learnt values to the old ones.
 - With $\alpha = 0.1$, we are assuring that the new value has some importance, as it is the updating value rate, and that the actual Q-value has a high ratio as it stores the information gathered after many iterations.
 - We give a weight of 0.9 to the old Q-value and 0.1 to the new reward.
- **Q-function**
 - First of all, we load the Q-values from a txt file. If there is no such file, the Q-values are initialized at 0.
 - **If training, update Q-function**
 - Select optimal action at state s with probability $1 - \epsilon$ or the other action with probability ϵ (off-policy).
 - At the end of an episode, update Q-values for all the visited state-action pairs.
 - At the end of the training, save the Q-values in a txt file.
 - **If testing, update the agent type**
 - Update agent 1 type (using previous positions and actual).
 - Select optimal action a at state s from the corresponding Q-function depending on the agent 1 type belief.
 - Store the results in a JSON file.

4.2.1 Q-learning training algorithm

We train both Q-functions (random and focused) with N episodes. The agent type is defined in this stage so the new reward will update the Q-values of the corresponding Q-functions on each visited state-action pairs.

First of all we initialize $Q(s,a)$ at 0 if there is no file with the Q-values stored. Then we enter the main loop (at 2), initialize R (that stores the sum of rewards of each episode) and the state s . Now we enter the inner loop (at 5), and while s is not terminal, we choose an action a at state s using the ϵ -greedy policy. Then we take the action and observe the reward and new state. When we reach a terminal state, we update the Q-

values (at 11) of all the visited state-action pairs at the actual episode using the learning rate and R (sum of rewards of the episode). After all the training episodes, we store the Q-values in a txt file (at 13). Take into consideration that we will train N episodes separately for each agent type and will store the Q-values in the corresponding txt file.

Training algorithm

```
1 Initialize  $Q(s,a)$  at 0
2 Initialize episode = 0
3 Initialize  $N\_EPISODES = 1000$ 
2 While episode <  $N\_EPISODES$  do
3     Initialize  $R=0$ 
4     Initialize state  $s$ 
5     While  $s$  is not terminal do
6         Choose  $a$  from  $s$  using  $\epsilon$ -greedy policy
7         Take action  $a$ , observe  $r, s'$ 
8          $R = R + r$ 
9          $s = s'$ 
10    For all visited  $(s,a)$  update  $Q$ 
11         $Q(s,a) = (1 - \alpha) * Q(s,a) + \alpha * R$ 
12    episode = episode + 1
13 Store  $Q(s,a)$  in a txt file
```

4.2.2 Q-learning testing algorithm

To test our Q-learning algorithm we run the agent over 100 episodes using the trained Q-functions. It is similar to the learning algorithm, but in this case, instead of updating the Q-values, we have to update the agent type belief. Then we look at the corresponding Q-function (random or focused) and choose the action that returns maximum expected utility at state s . Finally, we store the results in a JSON file. Later, we will use these results, in specific the mean score (per episode), to show the learning curve of our Q-learning algorithm.

Testing algorithm

```
1 Load both  $Q(s,a)$  from the txt files
2 Initialize episode = 0
3 Initialize  $N\_EPISODES = 100$ 
2 While episode <  $N\_EPISODES$  do
3     Initialize state  $s$ 
4     While  $s$  is not terminal do
5         Update agent type belief
6         Choose  $a$  from  $s$  using the learnt Q-function of the corresponding agent type
7         Take action  $a$ , observe  $r, s'$ 
8          $s = s'$ 
9     episode = episode + 1
10 Store the results in a json file
```

5. Experiments and results

To evaluate the efficiency and performance of our Q-learning algorithm, we can observe the evolution of the mean score (per episode) across a range of previously trained episodes. To do that we will use the stored results produced by the testing algorithm.

We will show a graph (*Figure 8*) with the mean score (per episode) after a training of N episodes, ranging between 0 and 2500, with a training step of 100 episodes. The mean score is made over 100 episodes testing.

Evaluating algorithm

```
1 Initialize TRAINING_STEP = 100
2 Initialize TOTAL_EPISODES = 2500
3 Initialize n_episodes = 0
4 While n_episodes < TOTAL_EPISODES do
5     Execute the testing algorithm
6     n_episodes = n_episodes + TRAINING_STEP
7     if n_episodes < TOTAL_EPISODES
8         Execute the training algorithm
```

On each iteration of the main loop (at 4), we execute the testing algorithm⁴ which uses the learnt Q-values to play the game and stores the results (in specific the mean score per episode) in a JSON file. Then we execute the training algorithm which allows the agent to learn for n_episodes against a Random agent and another n_episodes against a Focused one, updating the corresponding Q-values. The first testing is on an empty Q-function, without any previous learning, as it goes before the training. This way we can see the performance at 0 training games. On the last step, we skip the training, as we won't test it on the next iteration.



Figure 8. A chart of the mean score (per episode) across a range of trained episodes.

⁴ It is recommended to apply the learning and testing algorithm manually one after the other, as sometimes the game crashes.

At *Figure 8* we can see an improvement of the agent after 500 games and that it is heading up in many of the posterior trained games. However, we notice that there are some low mean scores at 600, 800 and 1200. As the generation of the environment is random, and so is the behaviour of the agent, there can be some games with lower scores even if the agent has trained for some games.

There can be many reasons for this low scoring. With a reward of only 15 points by catching the pig, agent might find a better solution in heading to the exit instead of chasing across the board, or it might try to chase the pig while the other agent isn't collaborating, losing points in its way, or the pig might be trapped in an uncatchable position, while agents make actions without any further reward. If some of these situations are faced multiple times in the testing, the results of the mean score can get even lower than the random behaviour, as we have seen.

Even with these problems, we can see that the score keeps improving slowly but constantly with the number of trained episodes, reaching a steady point after 1.400 trained episodes, where it stays above 5 with some 6 scorings.

Other interesting results

We have also saved a file with the number of times each state-action has been visited (see *Table 1*). After a training of 1.700 episodes, we find, for the focused Q-function, that a total of 7.865 updates have been made. Therefore, the mean of steps per episode against a focused agent is 4,62. For the random agent, a total of 10.769 Q-values have been updated so the mean steps per episode is 6,33.

Table 1. Number of visited state-actions in Q-values of random and focused functions.

Condition	Q-Focused updated times	Q-Random updated times
= 0 (not visited)	10.333	9.941
> 0 (total visited)	917	1.309
> 5	254	498
> 10	149	312
> 20	56	75
Max. visited	155	110
Total updates	7.865	10.769

With that values, we can say that both of the following are happening:

- Episodes end faster when the other agent is focused. It has completely sense as our agent will try to catch the pig whenever is possible and both agents would be heading into the pig.
- Against a random agent, our agent sometimes takes longer to model the other agent behaviour. That is completely normal as at the start we initially think that the agent is focused with a probability of 75% and act consequently. The other agent might even take some "focused" actions while being random, tricking our agent into making more moves to try to catch the pig.

From *Table 1* results, we observe that only 8% and 11% of the state-action pairs have been visited respectively in focused and random Q-function. Because of that, many random factors happening in the same testing can alter the results, giving a bad modelling of the learning curve at that specific point.

Using the same values of the training of 1.700 episodes we have the next results. A table with the number of occurrences of each specified condition in each type of Q-function: random and focused.

Table 2. Number of Q-values that accomplish the specified condition

Condition	Q-Focused values	Q-Random values
> 0	352	259
> 4	66	28
> 6	33	1
Max. value	11,9	7,02
< 0	548	1.038
< -2	47	322
< -4	0	28
Min. value	-3,87	-4,86

Looking at *Table 2*, we can see clearly that Q-Focused values are greater than Q-Random values on all the ranges: more positive values and less negative ones. Also, maximum and minimum values are intuitively greater on the Q-Focused function. Therefore, we can say that the expected value from the focused function is *usually* better than the random function on each state-action pair.

6. Conclusions

Finding an optimal policy in a collaborative multiagent systems is a complex task. We have used a simplified version of the Malmo Collaborative AI Challenge (MCAC) as our collaborative multiagent system, which includes many challenging problems like agent behaviour uncertainty (random or focused), limited learning trials (extremely time consuming) and complex interactions. We implemented a solution that is able to perform at great level in our version of the MCAC by using Q-learning alongside two solution approaches. To reduce the agent type uncertainty, we use an agent behaviour modelling so we can learn effective policies against each of the agent types (random and focused). For the limited learning trials problem, we propose a simplified version of the MCAC and a state-action abstraction: this way the executing time required for our agent to learn an optimal policy is reduced significantly. We reached very encouraging levels of improvement when training and testing our Q-learning implementation, proving the validity and effectiveness of Q-learning as a way to find optimal policies in collaborative multiagent systems where sequential decision making is needed.

7. References

[1] “Many problems such as (stochastic) planning problems, learning robot control and game playing problems have successfully been modelled in terms of an MDP. In fact MDPs have become the de facto standard formalism for learning sequential decision making.” Extract from [2]

[2] van Otterlo M., Wiering M. (2012) “Reinforcement Learning and Markov Decision Processes”. In: Wiering M., van Otterlo M. (eds) Reinforcement Learning. Adaptation, Learning, and Optimization, vol 12. Springer, Berlin, Heidelberg
https://link.springer.com/chapter/10.1007/978-3-642-27645-3_1

[3] “IBM's Deep Blue beats chess champion Garry Kasparov in 1997”
<http://www.nydailynews.com/news/world/kasparov-deep-blues-losingchess-champ-rooke-article-1.762264>

[4] Mnih, Volodymyr; et al. (December 2013). “Playing Atari with Deep Reinforcement Learning”. arXiv:1312.5602 [cs.LG]
<https://arxiv.org/abs/1312.5602>

[5] “The model is a convolutional neural network, trained with a variant of Q-learning, (...). We apply our method to seven Atari 2600 games from the Arcade Learning Environment, (...). We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.”
<https://arxiv.org/abs/1312.5602>

[6] David Silver; et al. (October 2017). “Mastering the game of Go without human knowledge”. Nature **volume 550**, pages 354–359

[7] “After just three days of self-play training, AlphaGo Zero emphatically defeated the previously [published version of AlphaGo](#) - which had itself [defeated 18-time world champion Lee Sedol](#) - by 100 games to 0. After 40 days of self training, AlphaGo Zero became even stronger, outperforming the version of AlphaGo known as “Master”, which has defeated the world's best players and [world number one Ke Jie](#).”
<https://deepmind.com/blog/alphago-zero-learning-scratch/>

[8] David Silver; et al. (December 2017). “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. arXiv:1712.01815 [cs.AI]
<https://arxiv.org/abs/1712.01815>

[9] “Starting from random play, and given no domain knowledge except the game rules, AlphaZero achieved within 24 hours a superhuman level of play in the games of chess and shogi (Japanese chess) as well as Go, and convincingly defeated a world-champion program in each case.”
<https://arxiv.org/abs/1712.01815>

[10] “We show that Q-learning converges to the optimum action-values with probability 1 (...)” Extract from [17]

[11] Johnson M., Hofmann K., Hutton T., Bignell D. (2016) [*The Malmo Platform for Artificial Intelligence Experimentation*](#). [*Proc. 25th International Joint Conference on Artificial Intelligence*](#), Ed. Kambhampati S., p. 4246. AAAI Press, Palo Alto, California USA. <https://github.com/Microsoft/malmo>

[12] Katja Hofmann; et al. (Early 2017). *The Malmo Collaborative AI Challenge*. <https://www.microsoft.com/en-us/research/academic-program/collaborative-ai-challenge/>
<https://github.com/Microsoft/malmo-challenge>
https://github.com/Microsoft/malmo-challenge/blob/master/ai_challenge/pig_chase/README.md

[13] Yoshida, W., Dolan, R.J. and Friston, K.J., (2008) [*Game Theory of Mind*](#). [*PLoS Computational Biology*](#), 4 (12), Article e1000254

[14] Russell, Stuart; Norvig, Peter (Original 1995). “Artificial Intelligence: A Modern Approach”. 3rd edition. (p. 41-44)

[15] Yanhai Xiong; Haipeng Chen; et al. (December 2017). “HogRider: Champion Agent of Microsoft Malmo Collaborative AI Challenge”. Nanyang Technological University http://www.ntu.edu.sg/home/boan/papers/AAAI18_Malmo.pdf

[16] <https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861>

[17] Watkins, C., Dayan, P. “Technical note: Q-learning”. *Machine Learning* 8 (3) (1992) 279–292

[18] Sutton, R.S., Barto, A.G. “Reinforcement Learning: An Introduction”. MIT Press, Cambridge, MA (1998)

[19] Watkins, C. “Learning from Delayed Rewards”. PhD thesis, University of Cambridge, Cambridge, England (1989)

[20] Heidrich-Meisner, V. “Interview with Richard S. Sutton”. *Künstliche Intelligenz* 3 (2009) 41–43

[21] <https://xlinux.nist.gov/dads/HTML/manhattanDistance.html>

8. Annexes

8.1 Previous installation

Follow the steps specified at the Malmo Collaborative AI Challenge (MCAC) main webpage⁵. There is a tutorial on how to install Malmo project (install Malmo 0.21 as recommended). To use the original MCAC, keep reading the tutorial and download the project at the github repository. You can follow the instructions to successfully install and execute the MCAC game.

Some problems I faced on a Windows 64-bit system, during installation and executing, were the following:

- Error with new Malmo version
 - o Solution: Installing Malmo 0.21
- Builtin error
 - o Solution: “pip install future”.
- “Get-item property cast invalid”
 - o Problem with NetBeans registry “nomodify”: it should be 4 bytes instead of 8. Change it in the registry editor⁶.

8.2 Running the original Malmo Collaborative AI Challenge (MCAC)

To run the original MCAC, two clients of the Malmo project need to be initialized:

Steps

- Start two instances of the Malmo Client on ports 10000 and 10001.
 - o Go to Malmo folder, Minecraft and execute each on one terminal:
 launchClient -port 10000
 launchClient -port 10001
- Go to directory
 cd malmo-challenge/ai_challenge/pig_chase
- Run a base agent
 python pig_chase_human_vs_agent.py

8.3 My code

Download my code zip.

There are some modified python files, the following are the most important ones, that can be found in the *malmo/py/ai_challenge/pig_chase* directory:

- **MyCustomAgent.py**: with **MyFinalAgent** class, where the all the Q-learning is implemented.
 - o The Q-functions will be saved in the same directory under the name defined in the `__init__` method of MyFinalAgent.
 - o When training or testing, the Q-functions are loaded from the txt files.

⁵ <https://github.com/Microsoft/malmo-challenge>

⁶ https://netbeans.org/bugzilla/show_bug.cgi?id=251538

- **run_train_agent.py**: to train the agent.
 - o It loads the Q-values (if found) with the specified name on “MyFinalAgent” init method.
 - o The Q-values against each type of agent are updated and stored in txt files.
 - o It is defaulted to run for 100 episodes.
- **run_evaluation.py**: to test the agent.
 - o It loads the Q-values like the run_train_agent
 - o It is defaulted to test for 100 episodes
 - o The results are stored in a JSON file.

Q-values of previous trainings can be found in the same directory as txt files under the name:

- **qFocused***
- **qFocusedVisited***
- **qRandom***
- **qRandomVisited***

Where * is a sequence of characters like “_v4_2” that indicates the “version” of the training. It has all the accumulated training.

And results of previous testings can be found under the name:

- **experiment_vX_Y_ZEP_N**

Where X and Y indicates the “version” of the Q-functions it used for the testing (like “_v4_2”) and ZEP indicates the number of episodes that the Q-function has trained over (like “2300EP”) and N to indicate other following iteration with the same trained Q-function.

8.4 Running my agent

Run two instances of the Malmö Client, as explained before. Make sure both clients are listening to their respective ports. Then run one of the following in the *malmo-challenge/ai_challenge/pig_chase* directory:

- To train the agent:
 - python .\run_train_agent.py**
- To test it:
 - python .\run_evaluation.py**
- It is recommended to set a low number of training games if you want to observe the learning curve.